16 Publishing and deploying your application

This chapter covers

- Publishing an ASP.NET Core application
- Hosting an ASP.NET Core application in IIS
- Customizing the URLs for an ASP.NET Core app
- Optimizing client-side assets with bundling and minification

We've covered a vast amount of ground so far in this book. We've gone over the basic mechanics of building an ASP.NET Core application, such as configuring dependency injection, loading app settings, and building a middleware pipeline. We've looked at the UI side, using Razor templates and layouts to build an HTML response. And we've looked at higher-level abstractions, such as EF Core and ASP.NET Core Identity, that let you interact with a database and add users to your application. In this chapter we're taking a slightly different route. Instead of looking at ways to build bigger and better applications, we'll focus on what it means to deploy your application so that users can access it.

We'll start by looking again at the ASP.NET Core hosting model in section 16.1 and examining why you might want to host your application behind a reverse proxy instead of exposing your app directly to the internet. I'll show you the difference between running an ASP.NET Core app in development using `dotnet run` and publishing the app for use on a remote server. Finally, I'll describe some of the options available to you when deciding how and where to deploy your app.

In section 16.2, I'll show you how to deploy your app to one such option, a Windows server running IIS (Internet Information Services). This is a typical deployment scenario for many developers already familiar with ASP.NET, so it will act as a useful case study, but it's certainly not the only possibility. I won't go into all the technical details of configuring the venerable IIS system, but I'll show you the bare minimum required to get it up and running. If your focus is cross-platform development, then don't worry, I don't dwell on IIS for too long.

In section 16.3, I'll provide an introduction to hosting on Linux. You'll see how it differs from hosting applications on Windows, learn the changes you need to make to your apps, and find out about some gotchas to look out for. I'll describe how reverse proxies on Linux differ from IIS and point you to some resources you can use to configure your environments, rather than giving exhaustive instructions in this book.

If you're *not* hosting your application using IIS, you'll likely need to set the URL that your ASP.NET Core app is using when you deploy your application. In section 16.4 I'll show two approaches to this: using the special `ASPNETCORE_URLS` environment variable and using command-line arguments. Although generally not an issue during development, setting the correct URLs for your app is critical when you need to deploy it.

In the final section of this chapter, we'll look at a common optimization step used when deploying your application. Bundling and minification are used to reduce the number and size of requests that browsers must make to your app to fully load a page. I'll show you how to use a simple tool to create bundles when you build your application, and how to conditionally load these when in production to optimize your app's page size.

This chapter covers a relatively wide array of topics, all related to deploying your app. But before we get into the nitty-gritty, I'll go over the hosting model for ASP.NET Core so that we're on the same page. This is significantly different from the hosting model of the previous version of ASP.NET, so if you're coming from that background, it's best to try to forget what you know!

16.1 Understanding the ASP.NET Core hosting model

If you think back to chapter 1, you may remember that we discussed the hosting model of ASP.NET Core. ASP.NET Core applications are, essentially, console applications. They have a `static void Main` function that serves as the entry point for the application, like a standard .NET console app would.

What makes an app an ASP.NET Core app is that it runs a web server, typically Kestrel, inside the console app process. Kestrel provides the HTTP functionality to receive requests and return responses to clients. Kestrel passes any requests it receives to the body of your application to generate a response, as shown in figure 16.1. This

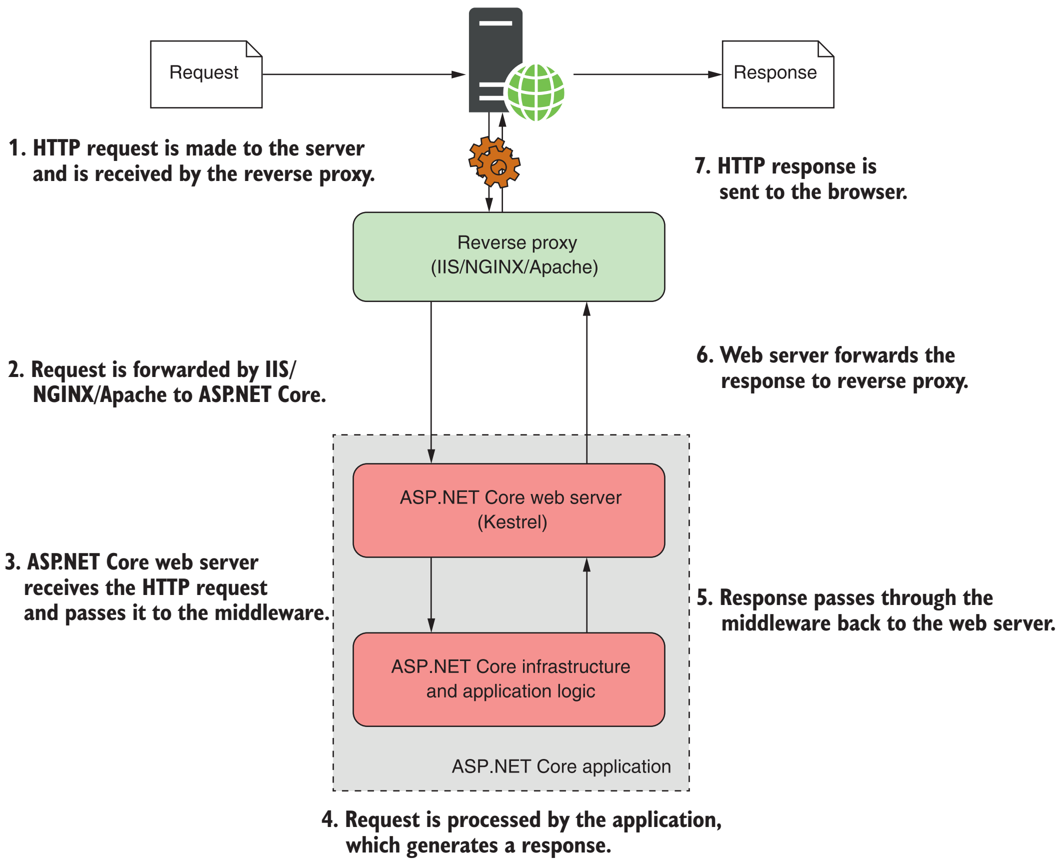

Request
Response
1. HTTP request is made to the server and is received by the reverse proxy.
7. HTTP response is sent to the browser.
Reverse proxy (IIS/NGINX/Apache)
2. Request is forwarded by IIS/NGINX/Apache to ASP.NET Core.
6. Web server forwards the response to reverse proxy.
ASP.NET Core web server (Kestrel)
3. ASP.NET Core web server receives the HTTP request and passes it to the middleware.
5. Response passes through the middleware back to the web server.
ASP.NET Core infrastructure and application logic
ASP.NET Core application
4. Request is processed by the application, which generates a response.

Figure 16.1 The hosting model for ASP.NET Core. Requests are received by the reverse proxy and are forwarded to the Kestrel web server. The same application can run behind various reverse proxies without modification.

hosting model decouples the server and reverse proxy from the application itself, so that the same application can run unchanged in multiple environments.

In this book we've focused on the lower half of figure 16.1—the ASP.NET Core application itself—but the reality is that you'll often want to place your ASP.NET Core apps behind a reverse proxy, such as IIS on Windows, or NGINX or Apache on Linux. The *reverse proxy* is the program that listens for HTTP requests from the internet and then makes requests to your app as though the request had come from the internet directly.

DEFINITION A *reverse proxy* is software that's responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy.

If you're running your application using a Platform as a Service (PaaS) offering, such as Azure App Service, you're using a reverse proxy there too—one that is managed by Azure. Using a reverse proxy has many benefits:

- *Security*—Reverse proxies are specifically designed to be exposed to malicious internet traffic, so they're typically well-tested and battle-hardened.
- *Performance*—You can configure reverse proxies to provide performance improvements by aggressively caching responses to requests.
- *Process management*—An unfortunate reality is that apps sometimes crash. Some reverse proxies can act as monitors/schedulers to ensure that if an app crashes, the proxy can automatically restart it.
- *Support for multiple apps*—It's common to have multiple apps running on a single server. Using a reverse proxy makes it easier to support this scenario by using the host name of a request to decide which app should receive the request.

I don't want to make it seem like using a reverse proxy is all sunshine and roses. There are some downsides:

- *Complexity*—One of the biggest complaints is how complex reverse proxies can be. If you're managing the proxy yourself (as opposed to relying on a PaaS implementation), there can be lots of proxy-specific pitfalls to look out for.
- *Inter-process communication*—Most reverse proxies require two processes: a reverse proxy and your web app. Communicating between the two is often slower than if you directly exposed your web app to requests from the internet.
- *Restricted features*—Not all reverse proxies support all the same features as an ASP.NET Core app. For example, Kestrel supports HTTP/2, but if your reverse proxy doesn't, you won't see the benefits.

Whether you choose to use a reverse proxy or not, when the time comes to host your app, you can't copy your code files directly to the server. First you need to *publish* your ASP.NET Core app, to optimize it for production. In section 16.1.1, we'll look at building an ASP.NET Core app so it can be run on your development machine, compared to publishing it so that it can be run on a server.

16.1.1 Running vs. publishing an ASP.NET Core app

One of the key changes in ASP.NET Core from previous versions of ASP.NET is making it easy to build apps using your favorite code editors and IDEs. Previously, Visual Studio was required for ASP.NET development, but with the .NET CLI and the OmniSharp plugin you can now build apps with the tools you're comfortable with, on any platform.

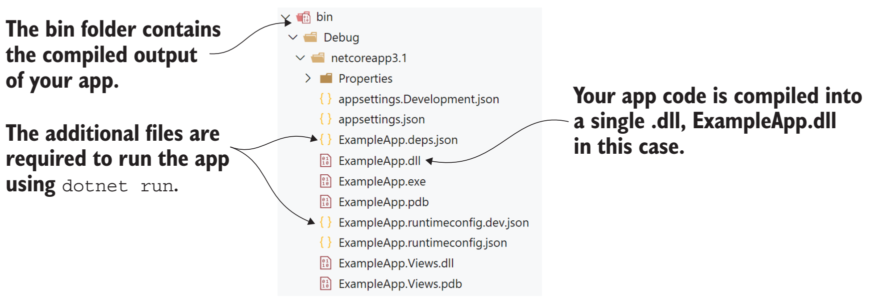
As a result, whether you build using Visual Studio or the .NET CLI, the same tools are being used under the hood. Visual Studio provides an additional GUI, functionality, and wrappers for building your app, but it executes the same commands as the .NET CLI behind the scenes.

As a refresher, you've used four main .NET CLI commands so far to build your apps:

- `dotnet new`—Creates an ASP.NET Core application from a template
- `dotnet restore`—Downloads and installs any referenced NuGet packages for your project
- `dotnet build`—Compiles and builds your project
- `dotnet run`—Executes your app, so you can send requests to it

If you've ever built a .NET application, whether it's an ASP.NET app or a .NET Framework console app, you'll know that the output of the build process is written to the bin folder. The same is true for ASP.NET Core applications.

If your project compiles successfully when you call `dotnet build`, the .NET CLI will write its output to a bin folder in your project's directory. Inside this bin folder are several files required to run your app, including a .dll file that contains the code for your application. Figure 16.2 shows part of the output of the bin folder for an ASP.NET Core application.



The bin folder contains the compiled output of your app.

The additional files are required to run the app using `dotnet run`.

Your app code is compiled into a single .dll, ExampleApp.dll in this case.

```
bin
  Debug
    netcoreapp3.1
      Properties
      appsettings.Development.json
      appsettings.json
      ExampleApp.deps.json
      ExampleApp.dll
      ExampleApp.exe
      ExampleApp.pdb
      ExampleApp.runtimeconfig.dev.json
      ExampleApp.runtimeconfig.json
      ExampleApp.Views.dll
      ExampleApp.Views.pdb
```

Figure 16.2 The bin folder for an ASP.NET Core app after running `dotnet build`. The application is compiled into a single .dll file, ExampleApp.dll.

> **NOTE** On Windows you will also have an executable .exe file, ExampleApp.exe. This is a simple wrapper file for convenience that makes it easier to run the application contained in ExampleApp.dll.

When you call `dotnet run` in your project folder (or run your application using Visual Studio), the .NET CLI uses the .dll to run your application. But this doesn't contain everything you need to deploy your app.

To host and deploy your app on a server, you first need to *publish* it. You can publish your ASP.NET Core app from the command line using the `dotnet publish` command. This builds and packages everything your app needs to run. The following command packages the app from the current directory and builds it to a subfolder called publish. I've used the `Release` configuration, instead of the default `Debug` configuration, so that the output will be fully optimized for running in production:

```
dotnet publish --output publish --configuration Release
```

TIP Always use the `release` configuration when publishing your app for deployment. This ensures the compiler generates optimized code for your app.

Once the command completes, you'll find your published application in the publish folder, as shown in figure 16.3.

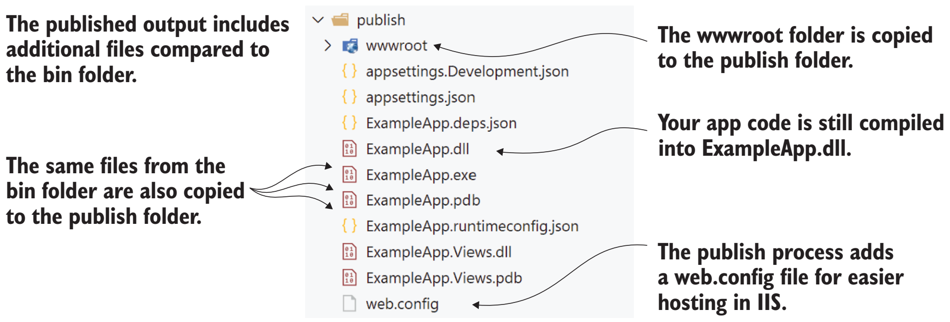

The published output includes additional files compared to the bin folder.
The same files from the bin folder are also copied to the publish folder.
publish
wwwroot
appsettings.Development.json
appsettings.json
ExampleApp.deps.json
ExampleApp.dll
ExampleApp.exe
ExampleApp.pdb
ExampleApp.runtimeconfig.json
ExampleApp.Views.dll
ExampleApp.Views.pdb
web.config
The wwwroot folder is copied to the publish folder.
Your app code is still compiled into ExampleApp.dll.
The publish process adds a web.config file for easier hosting in IIS.

Figure 16.3 The publish folder for the app after running `dotnet publish`. The app is still compiled into a single .dll file, but all the additional files, such as wwwroot and appsettings.json, are also copied to the output.

As you can see, the ExampleApp.dll file is still there, along with some additional files. Most notably, the publish process has copied across the wwwroot folder of static files. When running your application locally with `dotnet run`, the .NET CLI uses these files from your application's project folder directly. Running `dotnet publish` copies the files to the output directory, so they're included when you deploy your app to a server.

If your first instinct is to try running the application in the publish folder using the `dotnet run` command you already know and love, then you'll be disappointed. Instead of the application starting up, you'll be presented with a somewhat confusing message: "Couldn't find a project to run."

To run a published application, you need to use a slightly different command. Instead of calling `dotnet run`, you must pass the path to your application's .dll file to the `dotnet` command. If you're running the command from the publish folder, then for the example app in figure 16.3, that would look something like

```
dotnet ExampleApp.dll
```

This is the command that your server will run when running your application in production.

When you're developing, the `dotnet run` command does a whole load of work to make things easier on you: it makes sure your application is built, looks for a project file in the current folder, works out where the corresponding .dlls will be (in the bin folder), and finally, runs your app.

In production, you don't need any of this extra work. Your app is already built; it only needs to be run. The `dotnet <dll>` syntax does this alone, so your app starts much faster.

NOTE The `dotnet` command used to run your published application is part of the .NET Runtime, whereas the `dotnet` command used to build and run your application during development is part of the .NET SDK.

Framework-dependent deployments vs. self-contained deployments

.NET Core applications can be deployed in two different ways: runtime-dependent deployments (RDD) and self-contained deployments (SCD).

Most of the time, you'll use an RDD. This relies on the .NET 5.0 runtime being installed on the target machine that runs your published app, but you can run your app on any platform—Windows, Linux, or macOS—without having to recompile.

In contrast, an SCD contains *all* the code required to run your app, so the target machine doesn't need to have .NET 5.0 installed. Instead, publishing your app will package up the .NET 5.0 runtime with your app's code and libraries.

Each approach has its pros and cons, but in most cases I tend to create RDDs. The final size of RDDs is much smaller, as they only contain your app code, instead of the whole .NET 5.0 framework, which SCDs contain. Also, you can deploy your RDD apps to any platform, whereas SCDs must be compiled specifically for the target machine's operating system, such as Windows 10 64-bit or Red Hat Enterprise Linux 64bit.

That said, SCD deployments are excellent for isolating your application from dependencies on the hosting machine. SCD deployments don't rely on the version of .NET installed on a hosting provider, so you can, for example, use preview versions of .NET in Azure App Service without needing the preview version to be supported.

In this book, I only discuss RDDs for simplicity, but if you want to create an SCD, just provide a runtime identifier, in this case Windows 10 64-bit, when you publish your app:

```
dotnet publish -c Release -r win10-x64 -o publish_folder
```

The output will contain an .exe file, which is your application, and a *ton* of .dlls (about 65 MB of .dlls for a sample app), which are the .NET 5.0 framework. You need to deploy this whole folder to the target machine to run your app. In .NET 5.0 it's possible to trim some of these assemblies during the publish process, but this comes with risks in some scenarios. For more details, see Microsoft's ".NET Core application publishing overview" documentation at https://docs.microsoft.com/dotnet/core/deploying/.

We've established that publishing your app is important for preparing it to run in production, but how do you go about deploying it? How do you get the files from your computer onto a server so that people can access your app? You have many, many options for this, so in the next section I'll give you a brief list of approaches to consider.

16.1.2 Choosing a deployment method for your application

To deploy any application to production, you generally have two fundamental requirements:

- A server that can run your app
- A means of loading your app onto the server

Historically, putting an app into production was a laborious and error-prone process. For many people, this is still true. If you're working at a company that hasn't changed practices in recent years, you may need to request a server or virtual machine for your app and provide your application to an operations team who will install it for you. If that's the case, you may have your hands tied regarding how you deploy.

For those who have embraced continuous integration (CI) or continuous delivery/deployment (CD), there are many more possibilities. CI/CD is the process of detecting changes in your version control system (for example, Git, SVN, Mercurial, Team Foundation Version Control) and automatically building, and potentially deploying, your application to a server with little to no human intervention.[1]

You can find many different CI/CD systems out there: Azure DevOps, GitHub Actions, Jenkins, TeamCity, AppVeyor, Travis, and Octopus Deploy, to name a few. Each can manage some or all of the CI/CD process and can integrate with many different systems.

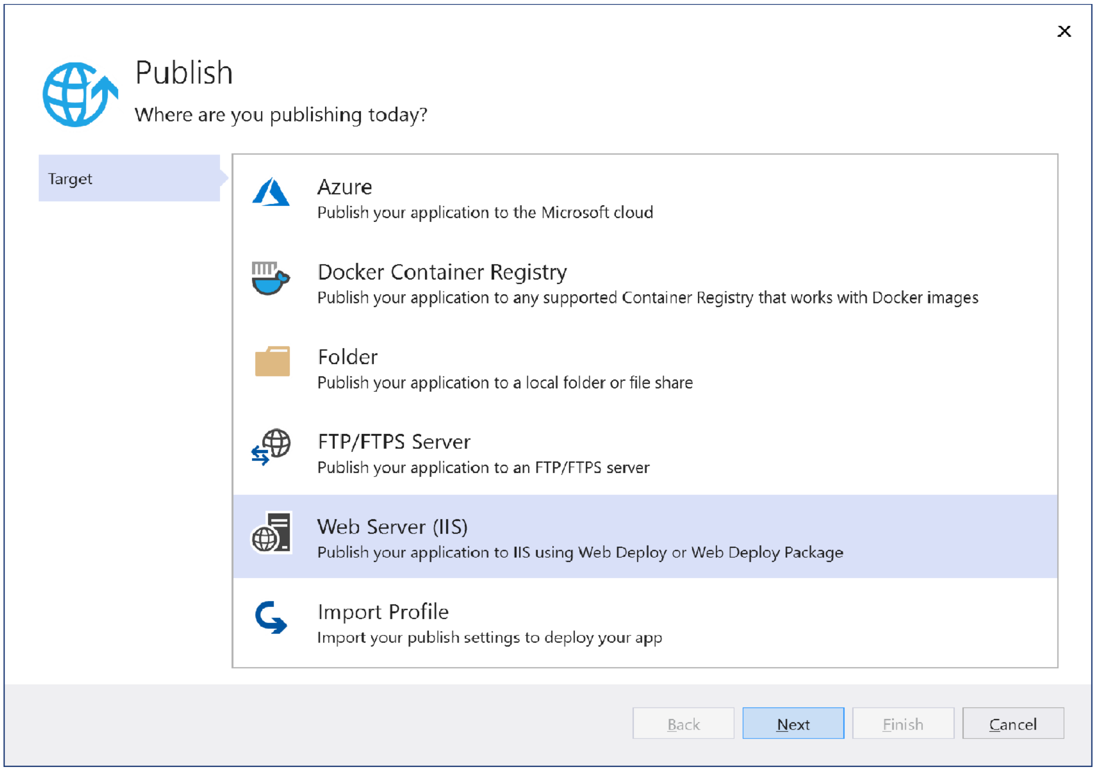
Rather than pushing any particular system, I suggest trying out some of the services available and seeing which works best for you. Some are better suited to open source projects, some are better when you're deploying to cloud services—it all depends on your particular situation.

If you're getting started with ASP.NET Core and don't want to have to go through the setup process of getting CI working, you still have lots of options. The easiest way to get started with Visual Studio is to use the built-in deployment options. These are available from Visual Studio via the Build > Publish *AppName* menu option, which presents you with the screen shown in figure 16.4.

From here, you can publish your application directly from Visual Studio to many different locations.[2] This is great when you're getting started, though I recommend looking at a more automated and controlled approach for larger applications, or when you have a whole team working on a single app.

[1] There are important but subtle differences between these terms. Atlassian has a good comparison article, "Continuous integration vs. continuous delivery vs. continuous deployment," here: http://mng.bz/vzp4.

[2] For guidance on choosing your Visual Studio publishing options, see Microsoft's "Deploy your app to a folder, IIS, Azure, or another destination" documentation: http://mng.bz/4Z8j.


Publish

Where are you publishing today?

Azure
Publish your application to the Microsoft cloud

Docker Container Registry
Publish your application to any supported Container Registry that works with Docker images

Folder
Publish your application to a local folder or file share

FTP/FTPS Server
Publish your application to an FTP/FTPS server

Web Server (IIS)
Publish your application to IIS using Web Deploy or Web Deploy Package

Import Profile
Import your publish settings to deploy your app

Figure 16.4 The Publish application screen in Visual Studio 2019. This provides easy options for publishing your application directly to Azure App Service, to IIS, to an FTP site, or to a folder on the local machine.

Given the number of possibilities available in this space, and the speed with which these options change, I'm going to focus on one specific scenario in this chapter: You've built an ASP.NET Core application and you need to deploy it. You have access to a Windows server that's already serving (previous version) ASP.NET applications using IIS, and you want to run your ASP.NET Core app alongside them.

In the next section, you'll see an overview of the steps required to run an ASP.NET Core application in production, using IIS as a reverse proxy. It won't be a master class in configuring IIS (there's so much depth to the 20-year-old product that I wouldn't know where to start!), but I'll cover the basics needed to get your application serving requests.

16.2 Publishing your app to IIS

In this section I'll briefly show you how to publish your first app to IIS. You'll add an application pool and website to IIS and ensure your app has the necessary configuration to work with IIS as a reverse proxy. The deployment itself will be as simple as copying your published app to IIS's hosting folder.

In section 16.1 you learned about the need to publish an app before you deploy it, and the benefits of using a reverse proxy when you run an ASP.NET Core app in production. If you're deploying your application to Windows, IIS will be your reverse proxy and will be responsible for managing your application.

IIS is an old and complex beast, and I can't possibly cover everything related to configuring it in this book. Nor would you want me to—it would be very boring! Instead, in this section I'll provide an overview of the basic requirements for running ASP.NET Core behind IIS, along with the changes you may need to make to your application to support IIS.

If you're on Windows and want to try out these steps locally, you'll need to manually enable IIS on your development machine. If you've done this with older versions of Windows, nothing much has changed. You can find a step-by-step guide to configuring IIS and troubleshooting tips in the ASP.NET Core documentation at http://mng.bz/6g2R.

16.2.1 Configuring IIS for ASP.NET Core

The first step in preparing IIS to host ASP.NET Core applications is to install the ASP.NET Core Windows Hosting Bundle.[3] This includes several components needed to run .NET apps:

- *The .NET Runtime*—Runs your .NET 5.0 application
- *The ASP.NET Core Runtime*—Required to run ASP.NET Core apps
- *The IIS AspNetCore Module*—Provides the link between IIS and your app, so that IIS can act as a reverse proxy

If you're going to be running IIS on your development machine, make sure you install the bundle as well, or you'll get strange errors from IIS.

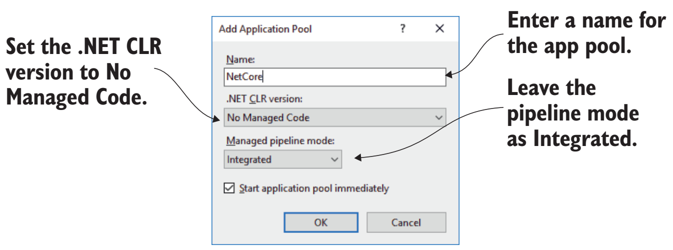
> **TIP** The Windows Hosting Bundle provides everything you need for running ASP.NET Core behind IIS on Windows. If you're hosting your application on Linux or Mac, or aren't using IIS on Windows, you'll need to install the .NET Runtime and ASP.NET Core Runtime to run runtime-dependent ASP.NET Core apps.

Once you've installed the bundle, you need to configure an *application pool* in IIS for your ASP.NET Core apps. Previous versions of ASP.NET would run in a *managed* app pool that used .NET Framework, but for ASP.NET Core you should create a *No Managed Code* pool. The native ASP.NET Core Module runs inside the pool, which boots the .NET 5.0 Runtime itself.

> **DEFINITION** An *application pool* in IIS represents an application process. You can run each app in IIS in a separate application pool to keep them isolated from one another.

[3] You can download the ASP.NET Core Windows Hosting Bundle from http://mng.bz/opED.

To create an unmanaged application pool, right-click Application Pools in IIS and choose Add Application Pool. Provide a name for the app pool in the resulting dialog box, such as NetCore, and set the .NET CLR version to No Managed Code, as shown in figure 16.5.

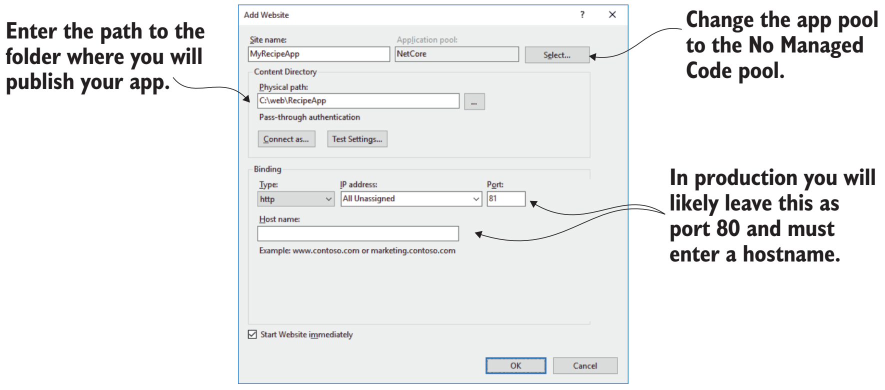

Add Application Pool
?
×
OK
Cancel
Set the .NET CLR version to No Managed Code.
Enter a name for the app pool.
Leave the pipeline mode as Integrated.

Name: NetCore
.NET CLR version: No Managed Code
Managed pipeline mode: Integrated
☑ Start application pool immediately

Figure 16.5 Creating an app pool in IIS for your ASP.NET Core app. The .NET CLR version should be set to No Managed Code.

Now that you have an app pool, you can add a new website to IIS. Right-click the Sites node and choose Add Website. In the Add Website dialog box, shown in figure 16.6, you provide a name for the website and the path to the folder where you'll publish your website. I've created a folder that I'll use to deploy the Recipe app from previous chapters. It's important to change the Application Pool for the app to the new Net-Core app pool you created. In production, you'd also provide a hostname for the application, but I've left it blank for now and changed the port to 81, so the application will bind to the URL http://localhost:81.

> **NOTE** When you deploy an application to production, you need to register a host name with a domain registrar, so your site is accessible by people on the internet. Use that hostname when configuring your application in IIS, as shown in figure 16.6.

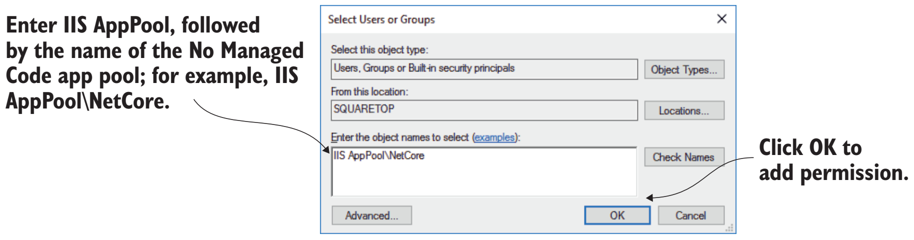

Add Website
?
×
Select...
Content Directory
...
Pass-through authentication
Connect as...
Test Settings...
Binding
Example: www.contoso.com or marketing.contoso.com
OK
Cancel
Enter the path to the folder where you will publish your app.
Change the app pool to the No Managed Code pool.
In production you will likely leave this as port 80 and must enter a hostname.

Site name: MyRecipeApp
Application pool: NetCore
Physical path: C:\web\RecipeApp
Type: http
IP address: All Unassigned
Port: 81
Host name:
☑ Start Website immediately

Figure 16.6 Adding a new website to IIS for your app. Be sure to change the Application Pool to the No Managed Code pool created in the previous step. You also provide a name, the path where you'll publish your app files, and the URL that IIS will use for your app.

Once you click OK, IIS creates the application and attempts to start it. But you haven't published your app to the folder, so you won't be able to view it in a browser yet.

You need to carry out one more critical setup step before you can publish and run your app: you must grant permissions for the NetCore app pool to access the path where you'll publish your app. To do this, right-click the folder that will host your app in Windows File Explorer and choose Properties. In the Properties dialog box, choose Security > Edit > Add. Enter `IIS AppPool\NetCore` in the text box, as shown in figure 16.7, where NetCore is the name of your app pool, and click OK. Close all the dialog boxes by clicking OK, and you're all set.


Enter IIS AppPool, followed by the name of the No Managed Code app pool; for example, IIS AppPool\NetCore.
Click OK to add permission.
Object Types...
Locations...
Check Names
Advanced...
OK
Cancel

Select Users or Groups

Select this object type:
Users, Groups or Built-in security principals

From this location:
SQUARETOP

Enter the object names to select (examples):
IIS AppPool\NetCore

Figure 16.7 Adding permission for the NetCore app pool to the website's publish folder.

Out of the box, the ASP.NET Core templates are configured to work seamlessly with IIS, but if you've created an app from scratch, you may need to make a couple of changes. In the next section I'll briefly show the changes you need to make and explain why they're necessary.

16.2.2 Preparing and publishing your application to IIS

As I discussed in section 16.1, IIS acts as a reverse proxy for your ASP.NET Core app. That means IIS needs to be able to communicate directly with your app to forward incoming requests to and outgoing responses from your app.

IIS handles this with the ASP.NET Core Module, but there's a certain degree of negotiation required between IIS and your app. For this to work correctly, you need to configure your app to use IIS integration.

IIS integration is added by default when you use the `IHostBuilder.Configure-WebHostDefaults()` helper method used in the default templates. If you're customizing your own `HostBuilder`, you need to ensure you add IIS integration with the `UseIIS()` or `UseIISIntegration()` extension method.

Listing 16.1 Adding IIS Integration to a host builder

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHost(webBuilder =>
            {
                webBuilder.UseKestrel();
                webBuilder.UseStartup<Startup>();
                webBuilder.UseIIS();
                webBuilder.UseIISIntegration();

            });
}
```

Using a custom builder, instead of the default ConfigureWebHostDefaults used in templates

Configures your application for use with IIS with an in-process hosting model

Configures your application for use with IIS with an out-of-process hosting model

NOTE If you're not using your application with IIS, the `UseIIS()` and `UseIIS-Integration()` methods will have no effect on your app, so it's safe to include them anyway.

In-process vs. out-of-process hosting in IIS

The reverse-proxy description I gave in section 16.1 assumes that your application is running in a separate process from the reverse proxy itself. That is the case if you're running on Linux and was the default for IIS up until ASP.NET Core 3.0.

In ASP.NET Core 3.0, ASP.NET Core switched to using an *in-process* hosting model by default for applications deployed to IIS. In this model, IIS hosts your application directly in the IIS process, reducing inter-process communication and boosting performance.

You can switch to the out-of-process hosting model with IIS if you wish, which can sometimes be useful for troubleshooting problems. Rick Strahl has an excellent post on the differences between the hosting models, how to switch between them, and the advantages of each: "ASP.NET Core In Process Hosting on IIS with ASP.NET Core" at http://mng.bz/QmEv.

In general, you shouldn't need to worry about the differences between the hosting models, but it's something to be aware of if you're deploying to IIS. If you choose to use the out-of-process hosting model, you should use the `UseIISIntegration()` extension method. If you use the in-process model, use `UseIIS()`. Alternatively, play it safe and use both—the correct extension method will be activated based on the hosting model used in production. Neither extension does anything if you don't use IIS.

When running behind IIS, these extension methods configure your app to pair with IIS so that it can seamlessly accept requests. Among other things, the extensions do the following:

- Define the URL that IIS will use to forward requests to your app and configure your app to listen on this URL
- Configure your app to interpret requests coming from IIS as coming from the client by setting up header forwarding
- Enable Windows authentication if required

Adding the IIS extension methods is the only change you need to make to your application to be able to host in IIS, but there's one additional aspect to be aware of when you publish your app.

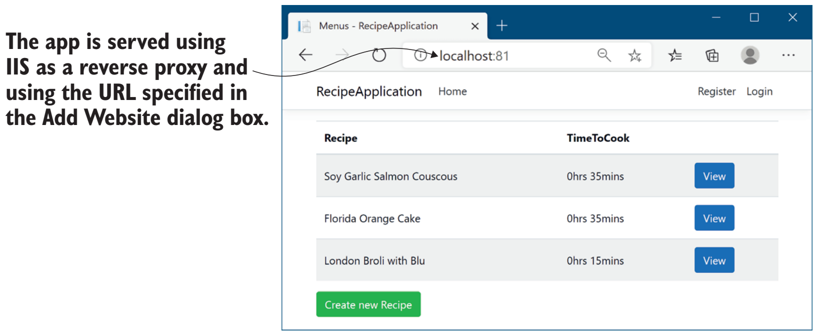
As with previous versions of ASP.NET, IIS relies on a web.config file to configure the applications it runs. It's important that your application include a web.config file when it's published to IIS; otherwise you could get broken behavior or even expose files that shouldn't be exposed.[4]

If your ASP.NET Core project already includes a web.config file, the .NET CLI or Visual Studio will copy it to the publish directory when you publish your app. If your app doesn't include a web.config file, the `publish` command will create the correct one for you. If you don't need to customize the web.config file, it's generally best not to include one in your project and let the CLI create the correct file for you.

With these changes, you're finally in a position to publish your application to IIS. Publish your ASP.NET Core app to a folder, either from Visual Studio or with the .NET CLI, by running

```
dotnet publish --output publish_folder --configuration Release
```

This will publish your application to the publish_folder folder. You can then copy your application to the path specified in IIS, as shown in figure 16.6. At this point, if all has gone smoothly, you should be able to navigate to the URL you specified for your app (http://localhost:81, in my case) and see it running, as shown in figure 16.8.


The app is served using IIS as a reverse proxy and using the URL specified in the Add Website dialog box.
Menus - RecipeApplication
localhost:81
RecipeApplication
Home
Register
Login
View
Create new Recipe

Recipe	TimeToCook	
Soy Garlic Salmon Couscous	0hrs 35mins	View
Florida Orange Cake	0hrs 35mins	View
London Broli with Blu	0hrs 15mins	View

Figure 16.8 The published application, using IIS as a reverse proxy listening at the URL http://localhost:81.

And there you have it, your first application running behind a reverse proxy. Even though ASP.NET Core uses a different hosting model than previous versions of ASP.NET, the process of configuring IIS is similar.

[4] For details on using web.config to customize the IIS AspNetCore Module, see Microsoft's "ASP.NET Core Module" documentation: http://mng.bz/Xdna.

As is often the case when it comes to deployment, the success you have is highly dependent on your precise environment and your app itself. If, after following these steps, you find you can't get your application to start, I highly recommend checking out the documentation at https://docs.microsoft.com/aspnet/core/publishing/iis. This contains many troubleshooting steps to get you back on track if IIS decides to throw a hissy fit.

This section was deliberately tailored to deploying to IIS, as it provides a great segue for developers who are already used to deploying ASP.NET apps and want to deploy their first ASP.NET Core app. But that's not to say that IIS is the only, or best, place to host your application.

In the next section I'll provide a brief introduction to hosting your app on Linux, behind a reverse proxy like NGINX or Apache. I won't go into configuration of the reverse proxy itself, but I will provide an overview of things to consider and resources you can use to run your applications on Linux.

16.3 Hosting an application on Linux

One of the great new features in ASP.NET Core is the ability to develop and deploy applications cross-platform, whether that be on Windows, Linux, or macOS. The ability to run on Linux, in particular, opens up the possibility of cheaper deployments to cloud hosting, deploying to small devices like a Raspberry Pi or to Docker containers.

One of the characteristics of Linux is that it's almost infinitely configurable. Although that's definitely a feature, it can also be extremely daunting, especially if you're coming from the Windows world of wizards and GUIs. This section provides an overview of what it takes to run an application on Linux. It focuses on the broad steps you need to take, rather than the somewhat tedious details of the configuration itself. Instead, I'll point to resources you can refer to as necessary.

16.3.1 Running an ASP.NET Core app behind a reverse proxy on Linux

You'll be glad to hear that running your application on Linux is broadly the same as running your application on Windows with IIS:

1. *Publish your app using* `dotnet publish`. If you're creating an RDD, the output is the same as you'd use with IIS. For an SCD, you must provide the target platform moniker, as described in section 16.1.1.
2. *Install the necessary prerequisites on the server.* For an RDD deployment, you must install the .NET 5.0 Runtime and the necessary prerequisites. You can find details on this in Microsoft's "Install .NET on Linux" documentation at https://docs.microsoft.com/en-gb/dotnet/core/install/linux.
3. *Copy your app to the server.* You can use any mechanism you like: FTP, USB stick, whatever you need to get your files onto the server!
4. *Configure a reverse proxy and point it to your app.* As you know by now, you may want to run your app behind a reverse proxy, for the reasons described in section 16.1.

On Windows, you'd use IIS, but on Linux you have more options. NGINX, Apache, and HAProxy are all commonly used options.

5. *Configure a process-management tool for your app.* On Windows, IIS acts both as a reverse proxy and a process manager, restarting your app if it crashes or stops responding. On Linux you typically need to configure a separate process manager to handle these duties; the reverse proxies won't do it for you.

The first three points are generally the same, whether you're running on Windows with IIS or on Linux, but the last two points are more interesting. In contrast to the monolithic IIS, Linux has a philosophy of small applications, each with a single responsibility.

IIS runs on the same server as your app and takes on multiple duties—proxying traffic from the internet to your app, but also monitoring the app process itself. If your app crashes or stops responding, IIS will restart the process to ensure you can keep handling requests.

In Linux, the reverse proxy might be running on the same server as your app, but it's also common for it to be running on a different server entirely, as shown in figure 16.9. This is similarly true if you choose to deploy your app to Docker; your app would typically be deployed in a container without a reverse proxy, and a reverse proxy on a server would point to your Docker container.

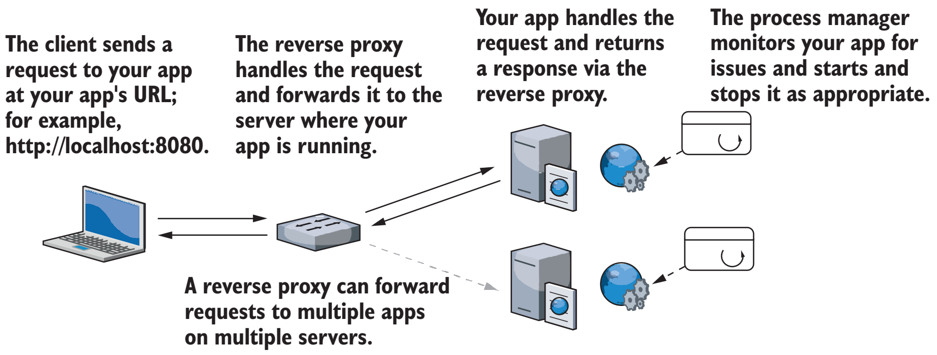

The client sends a request to your app at your app's URL; for example, http://localhost:8080.
The reverse proxy handles the request and forwards it to the server where your app is running.
Your app handles the request and returns a response via the reverse proxy.
The process manager monitors your app for issues and starts and stops it as appropriate.
A reverse proxy can forward requests to multiple apps on multiple servers.

Figure 16.9 On Linux it's common for a reverse proxy to be on a different server from your app. The reverse proxy forwards incoming requests to your app, while a process manager, such as systemd, monitors your apps for crashes and restarts it as appropriate.

As the reverse proxies aren't necessarily on the same server as your app, they can't be used to restart your app if it crashes. Instead, you need to use a process manager such as systemd to monitor your app. If you're using Docker, you'd typically use a container orchestrator such as Kubernetes (https://kubernetes.io) to monitor the health of your containers.

Running ASP.NET Core applications in Docker

Docker is the most commonly used engine for *containerizing* your applications. A container is like a small, lightweight virtual machine, specific to your app. It contains an operating system, your app, and any dependencies for your app. This container can then be run on any machine that runs Docker, and your app will run exactly the same, regardless of the host operating system and what's installed on it. This makes deployments highly repeatable: you can be confident that if the container runs on your machine, it will run on the server too.

ASP.NET Core is well-suited to container deployments, but moving to Docker involves a big shift in your deployment methodology and may or may not be right for you and your apps. If you're interested in the possibilities afforded by Docker and want to learn more, I suggest checking out the following resources:

- *Docker in Practice*, 2nd ed., by Ian Miell and Aidan Hobson Sayers (Manning, 2019) provides a vast array of practical techniques to help you get the most out of Docker (http://mng.bz/nM8d).
- Even if you're not deploying to Linux, you can use Docker with Docker for Windows. Check out the free e-book, *Introduction to Windows Containers* by John McCabe and Michael Friis (Microsoft Press, 2017) from https://aka.ms/containersebook.
- You can find a lot of details on building and running your ASP.NET Core applications on Docker in the .NET documentation at http://mng.bz/vz5a.
- Steve Gordon has an excellent blog post series on Docker for ASP.NET Core developers at https://www.stevejgordon.co.uk/docker-dotnet-developers.

Configuring these systems is a laborious task that makes for dry reading, so I won't detail them here. Instead, I recommend checking out the ASP.NET Core docs. They have a guide for NGINX and systemd, "Host ASP.NET Core on Linux with Nginx" (http://mng.bz/yYGd), and a guide for configuring Apache with systemd, "Host ASP.NET Core on Linux with Apache" (http://mng.bz/MXVB).

Both guides cover the basic configuration of the respective reverse proxies and systemd supervisors, but more importantly, they also show how to configure them *securely*. The reverse proxy sits between your app and the unfettered internet, so it's important to get it right!

Configuring the reverse proxy and the process manager is typically the most complex part of deploying to Linux, and that isn't specific to .NET development: the same would be true if you were deploying a Node.js web app. But you need to consider a few things *inside* your application when you're going to be deploying to Linux, as you'll see in the next section.

16.3.2 Preparing your app for deployment to Linux

Generally speaking, your app doesn't care which reverse proxy it sits behind, whether it's NGINX, Apache, or IIS—your app receives requests and responds to them without the reverse proxy affecting things. When you're hosting behind IIS, you need to add `UseIISIntegration()`; similarly, when you're hosting on Linux, you need to add an extension method to your app setup.

When a request arrives at the reverse proxy, it contains some information that is lost after the request is forwarded to your app. For example, the original request comes with the IP address of the client/browser connecting to your app: once the request is forwarded from the reverse proxy, the IP address is that of the *reverse proxy*, not the *browser*. Also, if the reverse proxy is used for SSL offloading (see chapter 18), then a request that was originally made using HTTPS may arrive at your app as an HTTP request.

The standard solution to these issues is for the reverse proxy to add additional headers before forwarding requests to your app. For example, the `X-Forwarded-For` header identifies the original client's IP address, while the `X-Forwarded-Proto` header indicates the original scheme of the request (`http` or `https`).

For your app to behave correctly, it needs to look for these headers in incoming requests and modify the request as appropriate. A request to http://localhost with the `X-Forwarded-Proto` header set to `https` should be treated the same as if the request was to https://localhost.

You can use `ForwardedHeadersMiddleware` in your middleware pipeline to achieve this. This middleware overrides `Request.Scheme` and other properties on `HttpContext` to correspond to the forwarded headers. If you're using the default `Host.CreateDefaultBuilder()` method in Program.cs, this is partially handled for you—the middleware is automatically added to the pipeline in a disabled state. To enable it, set the environment variable `ASPNETCORE_FORWARDEDHEADERS_ENABLED=true`.

If you're using your own `HostBuilder` instance, instead of the default builder, you can add the middleware to the start of your middleware pipeline manually, as shown in listing 16.2, and configure it with the headers to look for.

> **WARNING** It's important that `ForwardedHeadersMiddleware` be placed early in the middleware pipeline to correct `Request.Scheme` before any middleware that depends on the scheme runs.

Listing 16.2 Configuring an app to use forwarded headers in Startup.cs

```
public class Startup
{
    public class Configure(IApplicationBuilder app)
    {
        app.UseForwardedHeaders(new ForwardedHeadersOptions
        {
            ForwardedHeaders = ForwardedHeaders.XForwardedFor |
                               ForwardedHeaders.XForwardedProto
        });
```

Adds ForwardedHeadersMiddleware early in your pipeline

Configures the headers the middleware should look for and use

```
        app.UseHttpsRedirection();
        app.UseRouting();

        app.UseAuthentication();
        app.UseMvc();
    }
}
```

The forwarded headers middleware must be placed before all other middleware.

NOTE This behavior isn't specific to reverse proxies on Linux; the `UseIis()` extension adds `ForwardedHeadersMiddleware` under the hood as part of its configuration when your app is running behind IIS.

Aside from considering the forwarded headers, you need to consider a few minor things when deploying your app to Linux that may trip you up if you're used to deploying to Windows alone:

- *Line endings (`LF` on Linux versus `CRLF` on Windows)*—Windows and Linux use different character codes in text to indicate the end of a line. This isn't often an issue for ASP.NET Core apps, but if you're writing text files on one platform and reading them on a different platform, it's something to bear in mind.
- *Path directory separator (`"\"` on Windows, `"/"` on Linux)*—This is one of the most common bugs I see when Windows developers move to Linux. Each platform uses a different separator in file paths, so while loading a file using the `"subdir\myfile.json"` path will work fine on Windows, it won't on Linux. Instead, you should use `Path.Combine` to create the appropriate separator for the current platform; for example, `Path.Combine("subdir", "myfile.json")`.
- *Environment variables can't contain `":"`*—On some Linux distributions, the colon character `(:)` isn't allowed in environment variables. As you saw in chapter 11, this character is typically used to denote different sections in ASP.NET Core configuration, so you often need to use it in environment variables. Instead, you can use a double underscore in your environment variables (__) and ASP.NET Core will treat it the same as if you'd used a colon.
- *Time zone and culture data may be missing*—Linux distributions don't always come with time zone or culture data, which can cause localization issues and exceptions at runtime. You can install the time zone data using your distribution's package manager.[5]
- *Directory structures are different*—Linux distributions use a completely different folder structure than Windows, so you need to bear that in mind if your app hard-codes paths. In particular, consider differences in temporary/cache folders.

The preceding list is not exhaustive by any means, but as long as you set up `ForwardedHeadersMiddleware` and take care to use cross-platform constructs like `Path.Combine`,

[5] I ran into this problem myself. You can read about the issue in detail and how I solved it on my blog: http://mng.bz/aoem.

you shouldn't have too many problems running your applications on Linux. But configuring a reverse proxy isn't the simplest of activities, so wherever you're planning on hosting your app, I suggest checking the documentation for guidance at https://docs.microsoft.com/aspnet/core/publishing.

16.4 Configuring the URLs for your application

At this point, you've deployed an application, but there's one aspect you haven't configured: the URLs for your application. When you're using IIS as a reverse proxy, you don't have to worry about this inside your app. IIS integration with the ASP.NET Core Module works by dynamically creating a URL that's used to forward requests between IIS and your app. The hostname you configure in IIS (in figure 16.6) is the URL that external users see for your app; the internal URL that IIS uses when forwarding requests is never exposed.

If you're not using IIS as a reverse proxy—maybe you're using NGINX or exposing your app directly to the internet—you may find you need to configure the URLs your application listens to directly.

By default, ASP.NET Core will listen for requests on the URL http://localhost:5000. There are lots of ways to set this URL, but in this section I'll describe two: using environment variables or using command-line arguments. These are the two most common approaches I see (outside of IIS) for controlling which URLs your app uses.

TIP For further ways to set your application's URL, see my "5 ways to set the URLs for an ASP.NET Core app" blog entry: http://mng.bz/go0v.

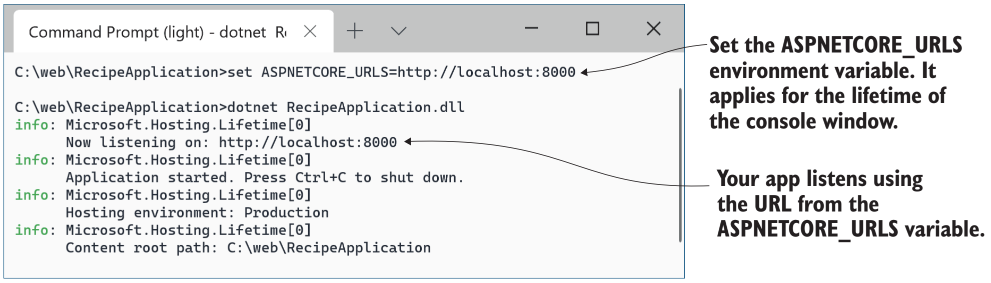
In chapter 10 you learned about configuration in ASP.NET Core, and in particular about the concept of hosting environments so that you can use different settings when running in development compared to production. You choose the hosting environment by setting an environment variable on your machine called `ASPNETCORE_ENVIRONMENT`. The ASP.NET Core framework magically picks up this variable when your app starts and uses it to set the hosting environment.

You can use a similar special environment variable to specify the URL that your app uses; this variable is called `ASPNETCORE_URLS`. When your app starts up, it looks for this value and uses it as the application's URL. By changing this value, you can change the default URL used by all ASP.NET Core apps on the machine.

For example, you could set a temporary environment variable in Windows from the command window using

```
set ASPNETCORE_URLS=http://localhost:8000
```

Running a published application using `dotnet <app.dll>` within the same command window, as shown in figure 16.10, shows that the app is now listening on the URL provided in the `ASPNETCORE_URLS` variable.


Command Prompt (light) - dotnet R
Set the ASPNETCORE_URLS environment variable. It applies for the lifetime of the console window.
Your app listens using the URL from the ASPNETCORE_URLS variable.

```
C:\web\RecipeApplication>set ASPNETCORE_URLS=http://localhost:8000

C:\web\RecipeApplication>dotnet RecipeApplication.dll
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:8000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\web\RecipeApplication
```

Figure 16.10 Change the `ASPNETCORE_URLS` environment variable to change the URL used by ASP.NET Core apps.

You can instruct an app to listen on multiple URLs by separating them with a semicolon, or you can listen to a specific port without specifying the localhost hostname. If you set the `ASPNETCORE_URLS` environment variable to

```
http://localhost:5001;http://*:5002
```

your ASP.NET Core apps will listen for requests sent to the following:

- http://localhost:5001—This address is only accessible on your local computer, so it will not accept requests from the wider internet.
- http://*:5002—Any URL on port 5002. External requests from the internet can access the app on port 5002, using any URL that maps to your computer.

Note that you *can't* specify a different hostname, like tastyrecipes.com, for example. ASP.NET Core will listen to all requests on a given port. The exception is the localhost hostname, which only allows requests that came from your own computer.

NOTE If you find the `ASPNETCORE_URLS` variable isn't working properly, ensure you don't have a launchSettings.json file in the directory. When present, the values in this file take precedence. By default, launchSettings.json isn't included in the `publish` output, so this generally won't be an issue in production.

Setting the URL of an app using a single environment variable works great for some scenarios, most notably when you're running a single application in a virtual machine, or within a Docker container.[6]

If you're not using Docker containers, the chances are you're hosting multiple apps side-by-side on the same machine. A single environment variable is no good for setting URLs in this case, as it would change the URL of every app.

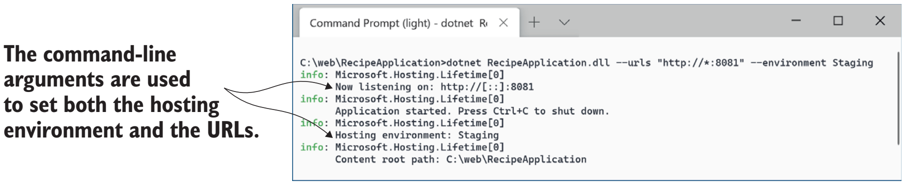
[6] ASP.NET Core is well-suited to running in containers, but working with containers is a separate book in its own right. For details on hosting and publishing apps using Docker, see Microsoft's "Host ASP.NET Core in Docker containers" documentation: http://mng.bz/e5GV.

In chapter 11 you saw that you could set the hosting environment using the `ASPNETCORE_ENVIRONMENT` variable, but you could also set the environment using the `--environment` flag when calling `dotnet run`:

```
dotnet run --no-launch-profile --environment Staging
```

You can set the URLs for your application in a similar way, using the `--urls` parameter. Using command-line arguments enables you to have multiple ASP.NET Core applications running on the same machine, listening to different ports. For example, the following command would run the recipe application, set it to listen on port 8081, and set the environment to staging, as shown in figure 16.11:

```
dotnet RecipeApplication.dll --urls "http://*:8081" --environment Staging
```


The command-line arguments are used to set both the hosting environment and the URLs.
Command Prompt (light) - dotnet R

```
C:\web\RecipeApplication>dotnet RecipeApplication.dll --urls "http://*:8081" --environment Staging
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://[::]:8081
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Staging
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\web\RecipeApplication
```

Figure 16.11 Setting the hosting environment and URLs for an application using command-line arguments. The values passed at the command line override values provided from appSettings.json or environment variables.

Remember, you don't need to set your URLs in this way if you're using IIS as a reverse proxy; IIS integration handles this for you. Setting the URLs is only necessary when you're manually configuring the URL your app is listening on; for example, if you're using NGINX or are exposing Kestrel directly to clients.

> **WARNING** If you are running your ASP.NET Core application without a reverse proxy, you should use *host filtering* for security reasons, to ensure your app only responds to requests for hostnames you expect. For more details, see my "Adding host filtering to Kestrel in ASP.NET Core" blog entry: http://mng.bz/pVXK.

Continuing the theme of deployment-related tasks, in the next section we'll take a look at optimizing some of your client-side assets for production. If you're building a Web API, this isn't something you'll have to worry about in your ASP.NET Core app, but for traditional web apps it's worth considering.

16.5 Optimizing your client-side assets using BundlerMinifier

In this section we'll explore the performance of your ASP.NET Core application in terms of the number and size of requests. You'll see how to improve the performance of your app using bundling and minification but ensure your app is still easy to debug while you're building it. Finally, we'll look at a common technique for improving app performance in production: using a content delivery network (CDN).

Have you ever used a web app or opened a web page that seemed to take forever to load? Once you stray off the beaten track of Amazon, Google, or Microsoft, it's only a matter of time before you're stuck twiddling your thumbs while the web page slowly pops into place.

Next time this happens to you, open the browser developer tools (for example, press F12 in Edge or Chrome) and take a look at the number and size of the requests the web app is making. In many cases, a high number of requests generating large responses will be responsible for the slow loading of a web page.

We'll start by exploring the problem of performance by looking at a single page from your recipe application: the login page. This is a simple page, and it isn't inherently slow, but even this is sufficient to investigate the impact of request size.

As a user, when you click the Login button, the browser sends a request to /Identity/Account/Login. Your ASP.NET Core app executes the Login.cshtml Razor Page in the default UI, which executes a Razor template and returns the generated HTML in the response, as shown in figure 16.12. That's a single request-response—a single round-trip.

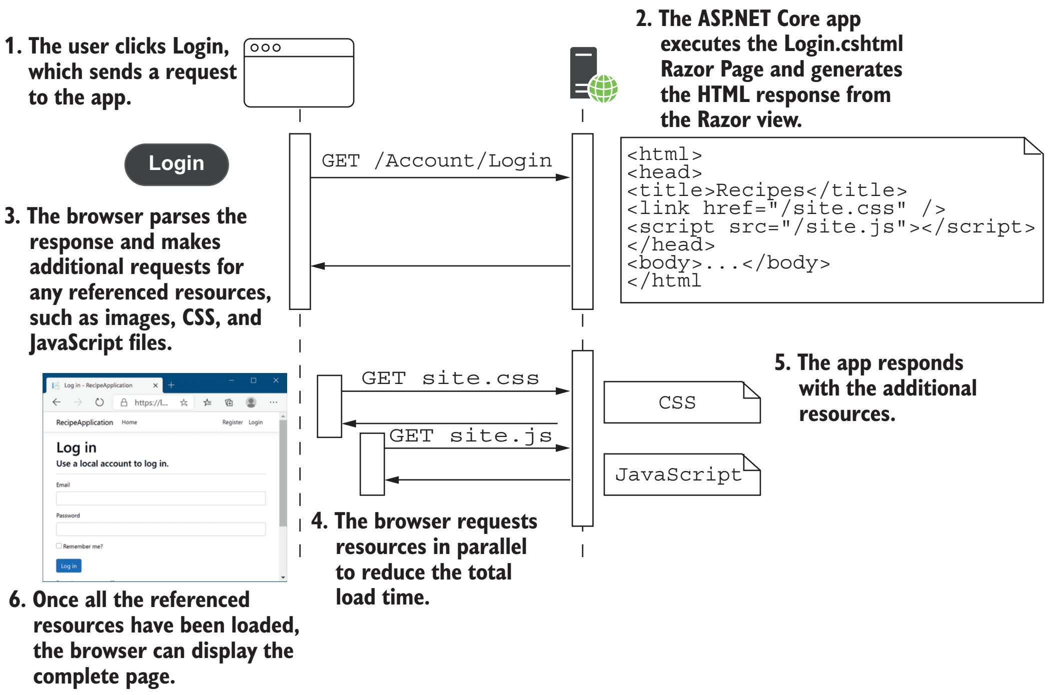
But that's not it for the web page. The HTML returned by the page includes links to CSS files (for styling the page), JavaScript files (for client-side functionality—client-side form validation, in this case), and, potentially, images and other resources (though you don't have any others in this recipe app).

The browser must request each of these additional files and wait for the server to return them before the whole page is loaded. When you're developing locally, this all happens quickly, as the request doesn't have far to go, but once you deploy your app to production, it's a different matter.

Users will be requesting pages from your app from a wide variety of distances from the server, and over a wide variety of network speeds. Suddenly, the number and size of the requests and responses for your app will have a massive impact on the overall perceived speed of your app. This, in turn, can have a significant impact on how users perceive your site, and, for e-commerce sites, even how much money they spend.[7]

A great way to explore how your app will behave for non-optimal networks is to use the network-throttling feature in Chrome's and Edge's developer tools. This simulates the delays and network speeds associated with different types of networks, so you can

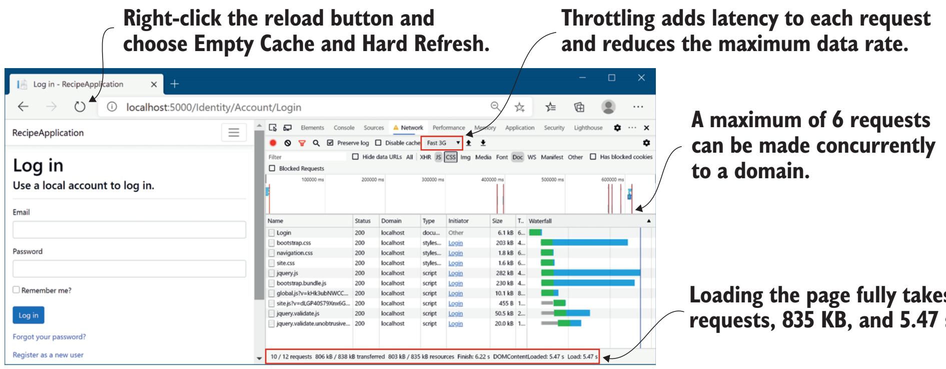
[7] There has been a lot of research done on this, including stats such as "a 0.1-second delay in page load time equals 7% loss in conversions" from http://mng.bz/4Z2Q.


1. The user clicks Login, which sends a request to the app.
Login
2. The ASP.NET Core app executes the Login.cshtml Razor Page and generates the HTML response from the Razor view.
GET /Account/Login
3. The browser parses the response and makes additional requests for any referenced resources, such as images, CSS, and JavaScript files.
GET site.css
GET site.js
CSS
JavaScript
4. The browser requests resources in parallel to reduce the total load time.
5. The app responds with the additional resources.
6. Once all the referenced resources have been loaded, the browser can display the complete page.
Log in
Use a local account to log in.
Email
Password
Remember me?
Log in

```
<html>
<head>
<title>Recipes</title>
<link href="/site.css" />
<script src="/site.js"></script>
</head>
<body>...</body>
</html
```

Figure 16.12 Loading a complete page for your app. The initial request returns the HTML for the page, but this may include links to other resources, such as CSS and JavaScript files. The browser must make additional requests to your app for all the outstanding resources before the page can be fully loaded.

get an idea of how your app behaves in the wild. In figure 16.13 I've loaded the login page for the recipe app, but this time with the network set to a modest Fast 3G speed.


Right-click the reload button and choose Empty Cache and Hard Refresh.
Throttling adds latency to each request and reduces the maximum data rate.
A maximum of 6 requests can be made concurrently to a domain.
Loading the page fully takes 10 requests, 835 KB, and 5.47 s.
Log in - RecipeApplication
localhost:5000/Identity/Account/Login
RecipeApplication
Log in
Use a local account to log in.
Email
Password
Remember me?
Log in
Forgot your password?
Register as a new user
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Preserve log
Disable cache
Fast 3G
Filter
Hide data URLs
All
XHR
JS
CSS
Img
Media
Font
Doc
WS
Manifest
Other
Has blocked cookies
Blocked Requests
100000 ms
200000 ms
300000 ms
400000 ms
500000 ms
600000 ms

Name	Status	Domain	Type	Initiator	Size	T...	Waterfall
Login	200	localhost	docu...	Other	6.1 kB	6...	
bootstrap.css	200	localhost	styles...	Login	203 kB	4...	
navigation.css	200	localhost	styles...	Login	1.8 kB	6...	
site.css	200	localhost	styles...	Login	1.6 kB	6...	
jquery.js	200	localhost	script	Login	282 kB	4...	
bootstrap.bundle.js	200	localhost	script	Login	230 kB	4...	
global.js?v=kHk3ubNWCC...	200	localhost	script	Login	10.1 kB	8...	
site.js?v=dLGP40S79Xnx6G...	200	localhost	script	Login	455 B	1...	
jquery.validate.js	200	localhost	script	Login	50.5 kB	2...	
jquery.validate.unobtrusive...	200	localhost	script	Login	20.0 kB	1...	

```
10 / 12 requests  806 kB / 838 kB transferred  803 kB / 835 kB resources  Finish: 6.22 s  DOMContentLoaded: 5.47 s  Load: 5.47 s
```

Figure 16.13 Exploring the effect of network speed on application load times. Chrome and Edge let you simulate a slower network, so you can get an impression of the experience users will have when loading your application once it's in production.

NOTE I've added additional files to the template—navigation.css and global.js—to make the page more representative of a real app.

Throttling the network doesn't change anything about the page or the data requested—there are 10 separate requests and almost 1 MB loaded for this single page—but it dramatically impacts the time for the page to load. Without throttling, the login page loads locally in 200 ms; with Fast 3G throttling, the login page takes 5.47 seconds to load!

NOTE Don't be too alarmed by these numbers. I'm making a point of reloading all the files with every request to emphasize the point, whereas in practice browsers go to great lengths to cache files to avoid having to send this amount of data.

The time it takes to fully load a page of your app is based primarily on two things:

- *The total size of the responses*—This is straight-up math; you can only return data at a certain speed, so the more data you need to return, the longer it takes.
- *The number of requests*—In general, the more requests the browser must make, the longer it takes to fully load the page. In HTTP/1.0 and HTTP/1.1, you can only make six concurrent requests to a server, so any requests after the sixth must wait for an earlier request to *finish* before they can even *start*. HTTP/2.0, which is supported by Kestrel, doesn't have this limit, but you can't always rely on clients using it.

How can you improve your app speed, assuming all the files you're serving are needed? In general, this is a big topic with lots of possibilities, such as using a CDN to serve your static files. Two of the simplest ways to improve your site's performance are to use *bundling* and *minification* to reduce the number and size of requests the browser must make to load your app.

16.5.1 Speeding up an app using bundling and minification

In figure 16.13 for the recipe app, you made a total of 10 requests to the server:

- One initial request for the HTML
- Three requests for CSS files
- Six requests for JavaScript files

Some of the CSS and JavaScript files are standard vendor libraries, like Bootstrap and jQuery, that are included as part of the default Razor templates, and some (navigation.css, site.css, global.js, and site.js) are files specific to your app. In this section, we're going to look at optimizing your custom CSS and JavaScript files.

If you're trying to reduce the number of requests for your app, an obvious first thought is to avoid creating multiple files in the first place. For example, instead of creating a navigation.css file and a site.css file, you could use a single file that contains all the CSS, instead of separating it out.

That's a valid solution, but putting all your code into one file may make it harder to manage and debug. As developers, we generally try to avoid this sort of monster file. A better solution is to split your code into as many files as you want, and then *bundle* the files when you come to deploy your code.

> **DEFINITION** *Bundling* is the process of concatenating multiple files into a single file, to reduce the number of requests.

Similarly, when you write JavaScript, you should use descriptive variables names, comments where necessary, and whitespace to create easily readable and debuggable code. When you come to deploy your scripts, you can process and optimize them for size, instead of readability. This process is called *minification*.

> **DEFINITION** *Minification* involves processing code to reduce its size without changing the behavior of the code. Processing has many different levels, which typically include removing comments and whitespace, and can extend to renaming variables to give them shorter names or removing whole sections of code entirely if they're unused.

As an example, look at the JavaScript in the following listing. This (very contrived) function adds up some numbers and returns them. It includes (excessively) descriptive variable names, comments, and plenty of use of whitespace, but it's representative of the JavaScript you might find in your own app.

Listing 16.3 Example JavaScript function before minification

```
function myFunc() {
    // this function doesn't really do anything,
    // it's just here so that we can show off minifying!
    function innerFunctionToAddTwoNumbers(
        thefirstnumber, theSecondNumber) {
        // i'm nested inside myFunc
        return thefirstnumber + theSecondNumber;
    }
    var shouldAddNumbers = true;
    var totalOfAllTheNumbers = 0;

    if (shouldAddNumbers == true) {
        for (var index = 0; i < 10; i++) {
            totalOfAllTheNumbers =
                innerFunctionToAddTwoNumbers(totalOfAllTheNumbers, index);
        }
    }
    return totalOfAllTheNumbers;
}
```

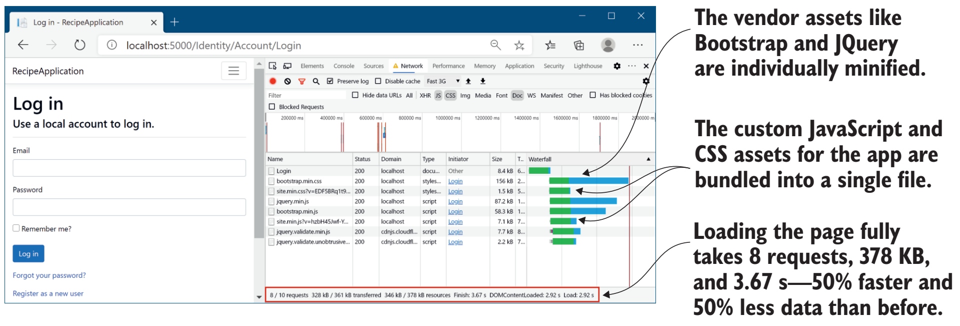
This function takes a total of 588 bytes as it's currently written, but after minification it's reduced to 95 bytes—16% of its original size. The behavior of the code is identical, but the output, shown in the following listing, is optimized to reduce its size. It's clearly not something you'd want to debug: you'd only use minified versions of your file in production; you'd use the original source files when developing.

Listing 16.4 Example JavaScript function after minification

```
function myFunc(){function r(n,t){return n+t}var
n=0,t;if(1)for(t=0;i<10;i++)n=r(n,t);return n}
```

Optimizing your static files using bundling and minification can provide a free boost to your app's performance when you deploy your app to production, while still letting you develop using easy-to-read and separated files.

Figure 16.14 shows the impact of bundling and minifying the files for the login page of the recipe app. Each of the vendor files has been minified to reduce its size, and the custom assets have been bundled and minified to reduce both their size and the number of requests. This reduced the number of requests from 10 to 8, the total amount of data from 580 KB to 270 KB, and the load time from 6.45 s to 3.15 s.


Log in - RecipeApplication
localhost:5000/Identity/Account/Login
RecipeApplication
Log in
Use a local account to log in.
Email
Password
Remember me?
Log in
Forgot your password?
Register as a new user
Elements Console Sources Network Performance Memory Application Security Lighthouse
Preserve log Disable cache Fast 3G
Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other Has blocked cookies
Blocked Requests
The vendor assets like Bootstrap and JQuery are individually minified.
The custom JavaScript and CSS assets for the app are bundled into a single file.
Loading the page fully takes 8 requests, 378 KB, and 3.67 s—50% faster and 50% less data than before.

Name	Status	Domain	Type	Initiator	Size	T..
Login	200	localhost	docu...	Other	8.4 kB	6...
bootstrap.min.css	200	localhost	styles...	Login	156 kB	2...
site.min.css?v=EDF5BRq1t9...	200	localhost	styles...	Login	1.5 kB	5...
jquery.min.js	200	localhost	script	Login	87.2 kB	1...
bootstrap.min.js	200	localhost	script	Login	58.3 kB	1...
site.min.js?v=hzbH45Jwf-Y...	200	localhost	script	Login	7.1 kB	7...
jquery.validate.min.js	200	cdnjs.cloudfl...	script	Login	7.7 kB	8...
jquery.validate.unobtrusive...	200	cdnjs.cloudfl...	script	Login	2.2 kB	7...

8 / 10 requests 328 kB / 361 kB transferred 346 kB / 378 kB resources Finish: 3.67 s DOMContentLoaded: 2.92 s Load: 2.92 s

Figure 16.14 By bundling and minifying your client-side resources, you can reduce both the number of requests required and the total data to transfer, which can significantly improve performance. In this example, bundling and minification cut the time to fully load in half.

> **NOTE** The vendor assets, such as jQuery and Bootstrap, aren't bundled with your custom scripts in figure 16.14. This lets you load those files from a CDN, as I'll touch on in section 16.5.4.

This performance improvement can be achieved with little effort on your part, and with no impact on your development process. In the next section I'll show how you can include bundling and minification as part of your ASP.NET Core build process, and how to customize the bundling and minification processes for your app.

16.5.2 Adding BundlerMinifier to your application

Bundling and minification isn't a new idea, so you have many ways to achieve the same result. The previous version of ASP.NET performed bundling and minification in managed code, whereas JavaScript task runners such as gulp, Grunt, and webpack are commonly used for these sorts of tasks. In fact, if you're writing a SPA, you're almost certainly already performing bundling and minification as a matter of course.

ASP.NET Core includes support for bundling and minification via a NuGet package called BuildBundlerMinifier or a Visual Studio extension version called Bundler & Minifier. You don't have to use either of these, and if you're already using other tools such as gulp or webpack, I suggest you continue to use them instead. But if you're getting started with a new project, I suggest considering BundlerMinifier; you can always switch to a different tool later.

You have two options for adding BundlerMinifier to your project:

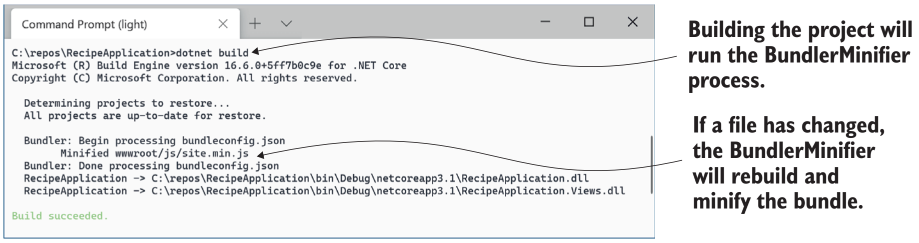
- You can install the Bundler & Minifier Visual Studio extension from Tools > Extensions and Updates.
- You can add the BuildBundlerMinifier NuGet package to your project.

Whichever approach you use, they both use the same underlying BundlerMinifier library.[8] I prefer to use the NuGet package approach, as it's cross-platform and will automatically bundle your resources for you, but the extension is useful for performing ad hoc bundling. If you do use the Visual Studio extension, make sure you enable Bundle on build, as you'll see shortly.

You can install the BuildBundlerMinifier NuGet package in your project with this command:

```
dotnet add package BuildBundlerMinifier
```

With the BuildBundlerMinifier package installed, whenever you build your project, the BundlerMinifier will check your CSS and JavaScript files for changes. If something has changed, new bundled and minified files are created, as shown in figure 16.15, where I modified a JavaScript file.



```
C:\repos\RecipeApplication>dotnet build
Microsoft (R) Build Engine version 16.6.0+5ff7b0c9e for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.

  Bundler: Begin processing bundleconfig.json
        Minified wwwroot/js/site.min.js
  Bundler: Done processing bundleconfig.json
  RecipeApplication -> C:\repos\RecipeApplication\bin\Debug\netcoreapp3.1\RecipeApplication.dll
  RecipeApplication -> C:\repos\RecipeApplication\bin\Debug\netcoreapp3.1\RecipeApplication.Views.dll

Build succeeded.
```

Command Prompt (light)
Building the project will run the BundlerMinifier process.
If a file has changed, the BundlerMinifier will rebuild and minify the bundle.

Figure 16.15 Whenever the project is built, the BuildBundlerMinifier tool looks for changes in the input files and builds new bundles as necessary.

[8] The BundlerMinifier library and extension are open source on GitHub: https://github.com/madskristensen/BundlerMinifier/.

As you can see in figure 16.15, the bundler minified your JavaScript code and created a new file at wwwroot/js/site.min.js. But why did it pick this name? Well, because you told it to in bundleconfig.json. You add this JSON file to the root folder of your project, and it controls the BundlerMinifier process.

Listing 16.5 shows a typical bundleconfig.json configuration for a small app. This defines which files to include in each bundle, where to write each bundle, and what minification options to use. Two bundles are configured here: one for CSS and one for JavaScript. You can add bundles to the JSON array, or you can customize the existing provided bundles.

Listing 16.5 A typical bundleconfig.json file

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
      "wwwroot/css/navigation.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
      "wwwroot/js/*.js",
      "!wwwroot/js/site.min.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

The bundler will create a file with this name.

The files listed in inputFiles are minified and concatenated into outputFileName.

You should create separate bundles for CSS and JavaScript.

You can specify multiple bundles, each with an output filename.

You can use globbing patterns to specify files to include.

The ! symbol excludes the matching file from the bundle.

The JavaScript bundler has some additional options.

You can optionally create a source map for the bundled JavaScript file.

For each bundle you can list a specific set of files to include, as specified in the `inputFiles` property. If you add a new CSS or JavaScript file to your project, you have to remember to come back to bundleconfig.json and add it to the list.

Alternatively, you can use *globbing* patterns to automatically include new files by default. This isn't always possible, such as when you need to ensure that files are concatenated in a given order, but I find it preferable in many cases. The example in the previous listing uses globbing for the JavaScript bundle: the pattern includes all .js files in the wwwroot/js folder but excludes the minified output file itself.

DEFINITION *Globbing* patterns use wildcards to represent many different characters. For example, *.css would match all files with a .css extension, whatever the filename.

When you build your project, the BundlerMinifier optimizes your CSS files into a single wwwroot/css/site.min.css file and your JavaScript into a single wwwroot/js/site .min.js file, based on the settings in bundleconfig.json. In the next section we'll look at how to include these files when you run in production, continuing to use the original files when developing locally.

> **NOTE** The BundlerMinifier package is great for optimizing your CSS and JavaScript resources. But images are another important resource to consider for optimization, and they can easily constitute the bulk of your page's size. Unfortunately, there aren't any built-in tools for this, so you'll need to look at other options; TinyPNG (https://tinypng.com) is one.

16.5.3 Using minified files in production with the Environment Tag Helper

Optimizing files using bundling and minification is great for performance, but you want to use the original files during development. The easiest way to achieve this split in ASP.NET Core is to use the Environment Tag Helper to conditionally include the original files when running in the development hosting environment, and to use the optimized files in other environments.

You learned about the Environment Tag Helper in chapter 8, where we used it to show a banner on the homepage when the app was running in the testing environment. Listing 16.6 shows how you can take a similar approach in the _Layout .cshtml page for the CSS files of your recipe app by using two Environment Tag Helpers: one for when you're in development, and one for when you aren't (you're in production or staging, for example). You can use similar Tag Helpers for the JavaScript files in the app.

Listing 16.6 Using Environment Tag Helpers to conditionally render optimized files

```
<environment include="Development">
    <link rel="stylesheet"
        href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/navigation.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
    <link rel="stylesheet"
        href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.min.css" />
</environment>
```

Only render these links when running in the Development environment.

The development version of Bootstrap

The development version of your styles

Only render these links when not in Development, such as in Staging or Production.

The minified version of Bootstrap

The bundled and minified version of your styles

When the app detects it isn't running in the development hosting environment (as you learned in chapter 11), it will switch to rendering links for the optimized files.

This gives you the best of both worlds: performance in production and ease of development.

The example in listing 16.6 also included a minified version of Bootstrap, even though you didn't configure this as part of the BundlerMinifier. It's common for CSS and JavaScript libraries like Bootstrap to include a pre-minified version of the file for you to use in production. For those that do, it's often better to exclude them from your bundling process, as this allows you to potentially serve the file from a public CDN.

16.5.4 Serving common files from a CDN

A public CDN is a website that hosts commonly used files, such as Bootstrap or jQuery, which you can reference from your own apps. They have several advantages:

- They're normally fast.
- They save your server from having to serve the file, saving bandwidth.
- Because the file is served from a different server, it doesn't count toward the six concurrent requests allowed to your server in HTTP/1.0 and HTTP/1.1.[9]
- Many different apps can reference the same file, so a user visiting your app may have already cached the file by visiting a different website, and they may not need to download it at all.

It's easy to use a CDN in principle: reference the CDN file instead of the file on your own server. Unfortunately, you need to cater for the fact that, like any server, CDNs can sometimes fail. If you don't have a fallback mechanism to load the file from a different location, such as your server, this can result in your app looking broken.

Luckily, ASP.NET Core includes several Tag Helpers to make working with CDNs and fallbacks easier. Listing 16.7 shows how you could update your CSS Environment Tag Helper to serve Bootstrap from a CDN when running in production, and to include a fallback. The fallback test creates a temporary HTML element and applies a Bootstrap style to it. If the element has the expected CSS properties, the fallback test passes, because Bootstrap must be loaded. If it doesn't have the required properties, Bootstrap will be loaded from the alternative, local link instead.

Listing 16.7 Serving Bootstrap CSS styles from a CDN with a local fallback

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/navigation.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
    <link rel="stylesheet"  href="https://stackpath.bootstrapcdn.com/
➥     bootstrap/4.3.1/css/bootstrap.min.css"
```

By default, Bootstrap is loaded from a CDN.

[9] This limit is not fixed in stone, but modern browsers all use the same limit. See Push Technology's "Browser connection limitations" article: http://mng.bz/OEWw.

```
        asp-fallback-test-class="sr-only"
        asp-fallback-test-property="position"
        asp-fallback-test-value="absolute"
        asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css" />

    <link rel="stylesheet" href="~/css/site.min.css" />
</environment>
```

The fallback test applies the sr-only class to an element ...

... and checks that the element has a CSS position of absolute. This indicates Bootstrap was loaded.

If the fallback check fails, the CDN must have failed, so Bootstrap is loaded from the local link.

Optimizing your static files is an important step to consider before you put your app into production, as it can have a significant impact on performance. Luckily, the BuildBundlerMinifier package makes it easy to optimize your CSS and JavaScript files. If you couple that with serving common files from a CDN, your app will be as speedy as possible for users in production.

That brings us to the end of this chapter on publishing your app. This last mile of app development, deploying an application to a server where users can access it, is a notoriously thorny issue. Publishing an ASP.NET Core application is easy enough, but the multitude of hosting options available makes providing concise steps for every situation difficult.

Whichever hosting option you choose, there's one critical topic that is often overlooked but is crucial for resolving issues quickly: logging. In the next chapter you'll learn about the logging abstractions in ASP.NET Core, and how you can use them to keep tabs on your app once it's in production.

Summary

- ASP.NET Core apps are console applications that self-host a web server. In production, you typically use a reverse proxy, which handles the initial request and passes it to your app. Reverse proxies can provide additional security, operations, and performance benefits, but they can also add complexity to your deployments.
- .NET has two parts: the .NET SDK (also known as the .NET CLI) and the .NET Runtime. When you're developing an application, you use the .NET CLI to restore, build, and run your application. Visual Studio uses the same .NET CLI commands from the IDE.
- When you want to deploy your app to production, you need to publish your application, using `dotnet publish`. This creates a folder containing your application as a DLL, along with all its dependencies.
- To run a published application, you don't need the .NET CLI because you won't be building the app. You only need the .NET Runtime to run a published app. You can run a published application using the `dotnet app.dll` command, where app.dll is the application .dll created by the `dotnet publish` command.

- To host ASP.NET Core applications in IIS, you must install the ASP.NET Core Module. This allows IIS to act as a reverse proxy for your ASP.NET Core app. You must also install the .NET Runtime and the ASP.NET Core Runtime, which are installed as part of the ASP.NET Core Windows Hosting Bundle.
- IIS can host ASP.NET Core applications using one of two modes: in-process and out-of-process. The out-of-process mode runs your application as a separate process, as is typical for most reverse proxies. The in-process mode runs your application as part of the IIS process. This has performance benefits, as no inter-process communication is required.
- To prepare your application for publishing to IIS with ASP.NET Core, ensure you call `UseIISIntegration()` and `UseIIS()` on `WebHostBuilder`. The `ConfigureWebHostDefaults` static helper method does this automatically.
- When you publish your application using the .NET CLI, a web.config file will be added to the output folder. It's important that this file be deployed with your application when publishing to IIS, as it defines how your application should be run.
- The URL that your app will listen on is specified by default using the environment variable `ASPNETCORE_URLS`. Setting this value will change the URL for all the apps on your machine. Alternatively, pass the `--urls` command-line argument when running your app; for example, `dotnet app.dll --urls http://localhost:80`.
- It's important to optimize your client-side assets to reduce the size and number of files that must be downloaded by a client's web browser when a page loads. You can achieve this by bundling and minifying your assets.
- You can use the BuildBundlerMinifier package to combine multiple JavaScript or CSS files together in a process called bundling. You can reduce the size of the files in a process called minification, in which unnecessary whitespace is removed and variables are renamed while preserving the function of the file.
- You can install a Visual Studio extension to control bundling and minification, or you can install the BuildBundlerMinifier package to automatically perform bundling and minification on each build of the project. Using the extension allows you to minify on an ad hoc basis, but using the NuGet package allows you to automate the process.
- The settings for bundling and minification are stored in the bundleconfig.json file, where you can define the different output bundle files and choose which files to include in the bundle. You can explicitly specify files, or you can use globbing patterns to include multiple files using wildcard characters. Globbing is typically easier and less error prone, but you will need to specify files explicitly if they must be bundled in a specific order.
- Use the Environment Tag Helper to conditionally render your bundles in production only. This lets you optimize for performance in production and readability in development.

- For common files shared by multiple apps, such as jQuery or Bootstrap, you can serve files from a CDN. These are websites that host the common files, so your app doesn't need to serve them itself.
- Use Link and Script Tag Helpers to check that the file has loaded correctly from the CDN. These can test that a file has been downloaded by a client and ensures that your server is used as a fallback should the CDN fail.